June-2020 version for STM's STM32CubeIDE toolchain (newlib 3.0).
See 2020 version for NXP MCUXpresso with newlib 2.5-3.1

Background

For reliability, smaller embedded systems typically don't use dynamic memory, thus avoiding associated problems of leakage, fragmentation, and resulting out-of-memory crashes. In Nadler & Associates smaller embedded projects (and large avionics products), we've historically used ***no*** runtime dynamic memory.

Some tool chains (and especially some features of C++, the topic of a separate article) **require dynamic memory internally, even if your application uses no free storage calls**. The newlib C-runtime library (used in many embedded tool chains) internally uses it's own malloc-family routines. newlib maintains some internal buffers and requires some support for thread-safety.

This article explains how to use newlib safely in a FreeRTOS project with GNU toolchain.
Warning: This article discusses newlib version 3.0. Later versions **may be** different...

STM CubeMX Users - Beware

Newlib 3.0 is the only runtime library distributed in STM's STM32CubeIDE development environment. You can select *standard* or *reduced* for each of C and C++ (4 possible combinations). As of July-2019 (and still in June 2020! Unbelievable!), **Cube-generated projects using FreeRTOS** do not properly support malloc/free/etc and family, nor general newlib RTL reentrancy. **Your application will corrupt memory if it calls malloc/free/etc:**

- **directly**
- **via a newlib C-RTL function called by your application (for example sprintf %f), or**
- **via STM-provided HAL-LL code (for example STM's USB stack)**

This document explains how to fix this (follow the instructions at the bottom of this page).

newlib requirements

GNU ARM Embedded Toolchain distributions include a non-polluting reduced-size runtime library called newlib (or newlib-nano for the smallest variant). Unfortunately, newlib internally uses free storage (malloc/free) in startling places within the C runtime library.
Thus, newlib free storage routines get dragged in and used unexpectedly.
Ooops: Not MISRA-compliant, if you believe in that stuff.

The most common functions that bite unsuspecting embedded developers are `sprintf` using `%f`, `dtoa`, `ftoa`, `rand`, and `strtok`. Here's a list of newlib functions with reentrancy support (and hence using malloc).

newlib requires platform-specific support to:

- provide thread-safety,
- switch thread contexts, and
- obtain memory to be doled out and managed by malloc/free/etc (`sbrk`).

Here's the detailed list of support functions required by newlib.
If this support is properly implemented, newlib works well in a threaded environment like FreeRTOS.
CubeMX does not properly set up these support functions for a FreeRTOS project.
See: Bug Report to STM regarding CubeMX FreeRTOS projects and the STM bug discussion below.

Example newlib use of malloc/malloc_r:
Placing a breakpoint within malloc_r (innermost malloc routine) shows the following:

- newlib 3.0 does **not** use malloc before main() is called (unlike earlier NXP/newlib 2.5 startup, which malloc'd 5k). Verified for all four newlib variants.
- With simple decimal output, sprintf does not call malloc
- For %f output, sprintf does 4 malloc totalling ~200 bytes (only the first time it is called per task/thread).
  %f also requires proper linker arguments for float support.
- printf or similar function expected to do IO (as opposed to string operation) allocates an IO control structure of 428 bytes.

Tip: How can I see who is calling malloc ?
The basic steps are:

- create a wrapper function malloc_r,
- add the wrapper options to the linker, and
- place a breakpoint in malloc_r.

A complete example with instructions is included in the code below.

newlib under the hood - concurrency
newlib maintains information it needs to support each separate context (thread/task/ISR) in a **reentrancy structure**. This includes things like a thread-specific errno, thread-specific pointers to allocated buffers, etc. The active reentrancy structure is pointed at by global pointer _impure_ptr, which initially points to a statically allocated structure instance. If you use no threading or tasks, and you don't use malloc/free/etc or any of the reentrancy-dependent functions in multiple execution contexts, nothing more is required to use newlib in your application (newlib will maintain its required info in the single static structure). If your application or any library you use requires malloc/free/etc or any of the reentrancy-dependent functions in multiple contexts, newlib requires:

- concurrency protection for malloc/free/etc. The free storage pool will be corrupted if multiple threads call into these functions concurrently! To prevent reentrant execution of malloc/free/etc routines newlib requires hook procedures __malloc_lock/unlock If an RTOS-based application does not replace newlib's **complete** internal malloc family (FreeRTOS does not), _malloc_lock/unlock must be provided for thread safety.
- multiple reentrancy structures (one per context), and a mechanism to create, initialize, and cleanup these structures, plus switching _impure_ptr to point at the correct reentrancy structure each time the context changes.

WTF? Why are newlib's dtoa, ftoa, and sprintf calling malloc???
To quote Steele and White, *Isn't it a pain when you ask a computer to divide 1.0 by 10.0 and it prints 0.0999999?* To summarize and over-simplify a very long story, in 1990 my friend Will Clinger published a seminal paper giving a solution to this problem, but Will's solution required arbitrary precision and hence storage. See Will's retrospective on his seminal 1990 paper and subsequent developments. David Gay published an improved implementation (based on Will's paper), which was extremely widely adopted. newlib uses David Gay's code for dtoa and ftoa (using malloc), and sprintf uses dtoa and ftoa. Keith Packard's picolibc implements the no-malloc precise Ryū algorithm. There has been some recent discussion on the newlib mailing list including details of Gay's algorithm and maybe incorporating no-malloc Ryū algorithm from picolibc into newlib.
Really now, aren't you sorry you asked me why?

FreeRTOS support for newlib

FreeRTOS provides support for newlib's context management. In `FreeRTOSconfig.h`, add:
#define configUSE_NEWLIB_REENTRANT 1 // Required for thread-safety of newlib sprintf, strtok, etc...
With this option FreeRTOS does the following (in task.c):

- For each task, allocate and initialize a newlib reentrancy structure in the task control block (TCB). This adds 96* bytes overhead per task (* size hugely dependant on newlib build options), plus anything else newlib allocates as needed.
- Each task switch, set _impure_ptr to point to the newly active task's reentrancy structure.
- On task destruction, clean up the reentrancy structure (help newlib free any associated memory).

Careful: Depending on how your supplier built newlib, and whether you are using nano, `configUSE_NEWLIB_REENTRANT` can chew up considerable memory! Check carefully if this is OK for your application!

FreeRTOS memory management
FreeRTOS internally uses its own memory management scheme and API (implemented by your choice of heap*xx*.c routines). Some FreeRTOS applications only use FreeRTOS-provided memory management. Unfortunately many libraries use the standard "C" `malloc`-family routines internally, including newlib itself (for example STM's USB stack LL and HAL routines). For these cases I've implemented the FreeRTOS memory management API on top of the "C" standard (using newlib) in the module heap_useNewlib.c below.

Bugs in projects generated by STM's CubeMX
Note as of spring 2020: ST has attempted to solve some of problems discussed below, but rather than just redistributing heap_useNewlib (as NXP and others have done), ST has tried to redo the implementation - and failed to do it correctly! Best you delete their unfortunate failed attempts and use proven working code provided here.

From earlier versions: There are a number of problems in projects generated with STM's CubeMX tool, which lead to memory corruption and other unfortunate happenings:

1. STM does not provide __malloc_lock/unlock. Hence memory management is not thread-safe.
2. ***In a stunning bit of coding malpractice, STM's USB stack calls `malloc` from within an ISR*** (see for example function `USBD_CDC_Init`). Normally thread-safety is provided by suspending task-switching during memory management. Because of this incompetent use of `malloc` inside an ISR, interrupts need to be suspended during memory management. Unfortunately this can greatly increase interrupt latency. *ToDo: Verify priority set by `taskENTER_CRITICAL_FROM_ISR` actually blocks USB interrupt.*
3. newlib's free storage requires an external implementation of sbrk to provide the heap storage then doled out by the malloc-family (see newlib sbrk requirements).
   The `sbrk` implementation provided by STM <u>cannot</u> work with FreeRTOS.
   Any `malloc` request after the scheduler starts will fail.
4. STM did not enable (in `FreeRTOSconfig.h`):
   #define configUSE_NEWLIB_REENTRANT 1 // Required for thread-safety of newlib sprintf, strtok, etc...

Using newlib safely with FreeRTOS - Possible Approaches

If you are certain your application does not use any newlib functions or any library that internally use the malloc-family and/or depend on thread-specific reentrant context, you could do nothing. But, are you ***really*** sure??? What about the libraries you use (for example STM's USB stack LL and HAL routines)? For some hints, see this list of newlib functions with reentrancy support. The only really safe way to preclude accidental use is to provide HCF stubs for every one of these functions. Alternatively you can provide a linker -wrap command for each forbidden function but don't implement the wrappers, so link fails if forbidden routines are ever referenced.
<u>You really need to be sure!</u>

To avoid using newlib's printf and dragging in newlib reentrancy components, there are many cut-down printf implementations available (that do <u>not</u> use malloc). Won't help if you're using `dtoa` or `strtok` or others! Example light-weight printf implementations:

- Mario Viara's light-weight printf (optional floating point and reentrancy), used in *MCU on Eclipse Processor Expert*.
- mpaland printf (optional floating point).

- printf-stdarg.c distributed in the FreeRTOS Lab TCPIP example; this one only uses stack storage but does not implement floating point.
- https://github.com/ksstech/support_common .

You must ensure you don't accidentally use any newlib facilities requiring reentrancy support and/or malloc-family in multiple tasks.

Another option is wrap newlib's malloc-family to use FreeRTOS free storage (ie heap_4.c), and specify newlib support for FreeRTOS. Tell the linker to wrap all newlib's malloc-family functions (using -Xlinker --wrap=malloc etc.), and provide a wrapper function that calls the FreeRTOS functions. I tried that, but newlib's printf family uses realloc, which is not supported in FreeRTOS heap implementations.

In the end (thanks to Richard Damon for encouraging this approach), I implemented the FreeRTOS memory API on top of newlib's malloc family, and provided all the hooks newlib's malloc family requires.

Using newlib safely with FreeRTOS - Recommended Solution Details

If your application needs a complete `malloc` family implementation, or you are using *any* newlib functions that require malloc (for example `printf` family or `strtok`), do the following (I've provided an implementation below):

- Implement the hooks required by newlib (`sbrk, __malloc_lock/unlock`).
  Make sure your linker file matches the `sbrk` implementation!
- Provide a heap implementation that implements the FreeRTOS memory API using the malloc family of newlib.

To use the implementation I've provided in your project:

- Exclude from all builds any current FreeRTOS heap implementation, typically something like: `Middlewares\Third_Party\FreeRTOS\Source\portable\MemMang\heap_4.c`
- Exclude from all builds any current sbrk implementation. In older versions of RubeMX, ST generated a stand-alone sbrk.c module. Later versions hide it in `syscalls.c` to obfuscate their messes. Get rid of it!
- Add the module `heap_useNewlib_ST.c` I've provided.
- If in multiple tasks your application needs a complete `snprintf` implementation, `strtok`, `dtoa>`, or other newlib functions requiring reentrancy support, or you're not ***really*** sure...
  Configure FreeRTOS for newlib support. In `FreeRTOSconfig.h`, add the line:
  #define configUSE_NEWLIB_REENTRANT 1
  In RubeMX FreeRTOS options, select configUSE_NEWLIB_REENTRANT + deselect "FW pack heap file".
- If are sure you're only going to access newlib's reentrant routines from a single FreeRTOS task and want to skip configUSE_NEWLIB_REENTRANT (with its attendent overhead)? Ensure malloc is never called except during startup and from the designated single task. Verify with a check in malloc_lock (sorry, I did not provide example code).

Note: Latest code and further documentation is on Github Hide Code...

heap_useNewlib.c

```
1  /**
2   * \file heap_useNewlib_ST.c
3   * \brief Wrappers required to use newlib malloc-family within FreeRTOS.
4   *
5   * \par Overview
6   * Route FreeRTOS memory management functions to newlib's malloc family.
7   * Thus newlib and FreeRTOS share memory-management routines and memory pool,
8   * and all newlib's internal memory-management requirements are supported.
9   *
```

```
 * \author Dave Nadler
 * \date 20-August-2019
 * \version 27-Jun-2020 Correct "FreeRTOS.h" capitalization, commentary
 * \version 24-Jun-2020 commentary only
 * \version 11-Sep-2019 malloc accounting, comments, newlib version check
 *
 * \see http://www.nadler.com/embedded/newlibAndFreeRTOS.html
 * \see https://sourceware.org/newlib/libc.html#Reentrancy
 * \see https://sourceware.org/newlib/libc.html#malloc
 * \see https://sourceware.org/newlib/libc.html#index-_005f_005fenv_005flock
 * \see https://sourceware.org/newlib/libc.html#index-_005f_005fmalloc_005flock
 * \see https://sourceforge.net/p/freertos/feature-requests/72/
 * \see http://www.billgatliff.com/newlib.html
 * \see http://wiki.osdev.org/Porting_Newlib
 * \see http://www.embecosm.com/appnotes/ean9/ean9-howto-newlib-1.0.html
 *
 *
 * \copyright
 * (c) Dave Nadler 2017-2020, All Rights Reserved.
 * Web:         http://www.nadler.com
 * email:       drn@nadler.com
 *
 * Redistribution and use in source and binary forms, with or without modificatio
 * are permitted provided that the following conditions are met:
 *
 * - Use or redistributions of source code must retain the above copyright notice
 *   this list of conditions, and the following disclaimer.
 *
 * - Use or redistributions of source code must retain ALL ORIGINAL COMMENTS, ANI
 *   ANY CHANGES MUST BE DOCUMENTED, INCLUDING:
 *   - Reason for change (purpose)
 *   - Functional change
 *   - Date and author contact
 *
 * - Redistributions in binary form must reproduce the above copyright notice, th
 *   list of conditions and the following disclaimer in the documentation and/or
 *   other materials provided with the distribution.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AN
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
 * WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
 * DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE F
 * ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
 * (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
 * LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON
 * ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
 * (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
 * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 */

// ================================================================================
// ========================================  Configuration  =======================
// These configuration symbols could be provided by from build...
#define STM_VERSION // Replace sane LD symbols with STM CubeMX's poor standard ex
#define ISR_STACK_LENGTH_BYTES (configISR_STACK_SIZE_WORDS*4) // bytes to reserve
// ========================================  Configuration  =======================
// ================================================================================


#include <stdlib.h> // maps to newlib...
#include <malloc.h> // mallinfo...
#include <errno.h>  // ENOMEM
#include <stdbool.h>
#include <stddef.h>

```

```
#include "newlib.h"
#if ((__NEWLIB__ == 2) && (__NEWLIB_MINOR__ < 5)) ||((__NEWLIB__ == 3) && (__NEWL
  #warning "This wrapper was verified for newlib versions 2.5 - 3.1; please ensu
#endif

#include "FreeRTOS.h" // defines public interface we're implementing here
#if !defined(configUSE_NEWLIB_REENTRANT) ||  (configUSE_NEWLIB_REENTRANT!=1)
  #warning "#define configUSE_NEWLIB_REENTRANT 1 // Required for thread-safety o
  // If you're *REALLY* sure you don't need FreeRTOS's newlib reentrancy support,
#endif
#include "task.h"

// ============================================================================
// External routines required by newlib's malloc (sbrk/_sbrk, __malloc_lock/unlo
// ============================================================================

// Simplistic sbrk implementations assume stack grows downwards from top of memo
// and heap grows upwards starting just after BSS.
// FreeRTOS normally allocates task stacks from a pool placed within BSS or DATA
// Thus within a FreeRTOS task, stack pointer is always below end of BSS.
// When using this module, stacks are allocated from malloc pool, still always p
// current unused heap area...

// Doesn't work with FreeRTOS: STM CubeMX 2018-2019 Incorrect Implementation
#if 0
    caddr_t _sbrk(int incr)
    {
        extern char end asm("end"); // From linker: lowest unused RAM address, ju
        static char *heap_end;
        char *prev_heap_end;
        if (heap_end == 0) heap_end = &end;
        prev_heap_end = heap_end;
        if (heap_end + incr > stack_ptr) // Fails here:  always true for FreeRTOS
        {
            errno = ENOMEM; // ...so first call inside a FreeRTOS task lands here
            return (caddr_t) -1;
        }
        heap_end += incr;
        return (caddr_t) prev_heap_end;
    }
#endif

register char * stack_ptr asm("sp");

#ifdef STM_VERSION // Use STM CubeMX LD symbols for heap+stack area
    // To avoid modifying STM LD file (and then having CubeMX trash it), use ava
    // Unfortunately STM does not provide standardized markers for RAM suitable
    // STM CubeMX-generated LD files provide the following symbols:
    //    end     /* aligned first word beyond BSS */
    //    _estack /* one word beyond end of "RAM" Ram type memory, for STM32F429
    // Kludge below uses CubeMX-generated symbols instead of sane LD definitions
    #define __HeapBase  end
    #define __HeapLimit _estack // In K64F LD this is already adjusted for ISR st
    static int heapBytesRemaining;
    // no DRN HEAP_SIZE symbol from LD... // that's (&__HeapLimit)-(&__HeapBase)
    uint32_t TotalHeapSize; // publish for diagnostic routines; filled in first
#else
    // Note: DRN's K64F LD provided: __StackTop (byte beyond end of memory), __St
    // __HeapLimit was already adjusted to be below reserved stack area.
    extern char HEAP_SIZE;  // make sure to define this symbol in linker LD comma
    static int heapBytesRemaining = (int)&HEAP_SIZE; // that's (&__HeapLimit)-(&
#endif


#ifdef MALLOCS_INSIDE_ISRs // STM code to avoid malloc within ISR (USB CDC stack)
```

```
    // We can't use vTaskSuspendAll() within an ISR.
    // STM's stunningly bad coding malpractice calls malloc within ISRs (for exar
    // So, we must just suspend/resume interrupts, lengthening max interrupt resp
    #define DRN_ENTER_CRITICAL_SECTION(_usis) { _usis = taskENTER_CRITICAL_FROM_I
    #define DRN_EXIT_CRITICAL_SECTION(_usis)  { taskEXIT_CRITICAL_FROM_ISR(_usis)
#else
    #define DRN_ENTER_CRITICAL_SECTION(_usis) vTaskSuspendAll(); // Note: safe to
    #define DRN_EXIT_CRITICAL_SECTION(_usis)  xTaskResumeAll();  // Note: safe to
#endif

#ifndef NDEBUG
    static int totalBytesProvidedBySBRK = 0;
#endif
extern char __HeapBase, __HeapLimit;  // symbols from linker LD command file

// Use of vTaskSuspendAll() in _sbrk_r() is normally redundant, as newlib malloc
// __malloc_lock before calling _sbrk_r(). Note vTaskSuspendAll/xTaskResumeAll su

//! _sbrk_r version supporting reentrant newlib (depends upon above symbols defir
void * _sbrk_r(struct _reent *pReent, int incr) {
    #ifdef MALLOCS_INSIDE_ISRs // block interrupts during free-storage use
      UBaseType_t usis; // saved interrupt status
    #endif
    static char *currentHeapEnd = &__HeapBase;
    #ifdef STM_VERSION // Use STM CubeMX LD symbols for heap
      if(TotalHeapSize==0) {
        TotalHeapSize = heapBytesRemaining = (int)((&__HeapLimit)-(&__HeapBase))-
      };
    #endif
    char* limit = (xTaskGetSchedulerState()==taskSCHEDULER_NOT_STARTED) ?
            stack_ptr   :  // Before scheduler is started, limit is stack pointer
            &__HeapLimit-ISR_STACK_LENGTH_BYTES;  // Once running, OK to reuse al
    DRN_ENTER_CRITICAL_SECTION(usis);
    if (currentHeapEnd + incr > limit) {
        // Ooops, no more memory available...
        #if( configUSE_MALLOC_FAILED_HOOK == 1 )
          {
            extern void vApplicationMallocFailedHook( void );
            DRN_EXIT_CRITICAL_SECTION(usis);
            vApplicationMallocFailedHook();
          }
        #elif defined(configHARD_STOP_ON_MALLOC_FAILURE)
            // If you want to alert debugger or halt...
            // WARNING: brkpt instruction may prevent watchdog operation...
            while(1) { __asm("bkpt #0"); }; // Stop in GUI as if at a breakpoint
        #else
            // Default, if you prefer to believe your application will gracefully
            pReent->_errno = ENOMEM; // newlib's thread-specific errno
            DRN_EXIT_CRITICAL_SECTION(usis);
        #endif
        return (char *)-1; // the malloc-family routine that called sbrk will ret
    }
    // 'incr' of memory is available: update accounting and return it.
    char *previousHeapEnd = currentHeapEnd;
    currentHeapEnd += incr;
    heapBytesRemaining -= incr;
    #ifndef NDEBUG
        totalBytesProvidedBySBRK += incr;
    #endif
    DRN_EXIT_CRITICAL_SECTION(usis);
    return (char *) previousHeapEnd;
}
//! non-reentrant sbrk uses is actually reentrant by using current context
// ... because the current _reent structure is pointed to by global _impure_ptr
char * sbrk(int incr) { return _sbrk_r(_impure_ptr, incr); }
```

```
205 //! _sbrk is a synonym for sbrk.
206 char * _sbrk(int incr) { return sbrk(incr); };
207
208 #ifdef MALLOCS_INSIDE_ISRs // block interrupts during free-storage use
209   static UBaseType_t malLock_uxSavedInterruptStatus;
210 #endif
211 void __malloc_lock(struct _reent *r)    {
212   #if defined(MALLOCS_INSIDE_ISRs)
213     DRN_ENTER_CRITICAL_SECTION(malLock_uxSavedInterruptStatus);
214   #else
215     bool insideAnISR = xPortIsInsideInterrupt();
216     configASSERT( !insideAnISR ); // Make damn sure no more mallocs inside ISRs!
217   vTaskSuspendAll();
218   #endif
219 };
220 void __malloc_unlock(struct _reent *r) {
221   #if defined(MALLOCS_INSIDE_ISRs)
222     DRN_EXIT_CRITICAL_SECTION(malLock_uxSavedInterruptStatus);
223   #else
224   (void)xTaskResumeAll();
225   #endif
226 };
227
228 // newlib also requires implementing locks for the application's environment memo
229 // accessed by newlib's setenv() and getenv() functions.
230 // As these are trivial functions, momentarily suspend task switching (rather tha
231 // Not required (and trimmed by linker) in applications not using environment var
232 // ToDo: Move __env_lock/unlock to a separate newlib helper file.
233 void __env_lock()    {       vTaskSuspendAll(); };
234 void __env_unlock()  { (void)xTaskResumeAll();  };
235
236 #if 1 // Provide malloc debug and accounting wrappers
237   /// /brief  Wrap malloc/malloc_r to help debug who requests memory and why.
238   /// To use these, add linker options: -Xlinker --wrap=malloc -Xlinker --wrap=_r
239   // Note: These functions are normally unused and stripped by linker.
240   size_t TotalMallocdBytes;
241   int MallocCallCnt;
242   static bool inside_malloc;
243   void *__wrap_malloc(size_t nbytes) {
244     extern void * __real_malloc(size_t nbytes);
245     MallocCallCnt++;
246     TotalMallocdBytes += nbytes;
247     inside_malloc = true;
248       void *p = __real_malloc(nbytes); // will call malloc_r...
249     inside_malloc = false;
250     return p;
251   };
252   void *__wrap__malloc_r(void *reent, size_t nbytes) {
253     extern void * __real__malloc_r(size_t nbytes);
254     if(!inside_malloc) {
255       MallocCallCnt++;
256       TotalMallocdBytes += nbytes;
257     };
258     void *p = __real__malloc_r(nbytes);
259     return p;
260   };
261 #endif
262
263 // ============================================================================
264 // Implement FreeRTOS's memory API using newlib-provided malloc family.
265 // ============================================================================
266
267 void *pvPortMalloc( size_t xSize ) PRIVILEGED_FUNCTION {
268     void *p = malloc(xSize);
269     return p;
```

```
}
void vPortFree( void *pv ) PRIVILEGED_FUNCTION {
    free(pv);
};

size_t xPortGetFreeHeapSize( void ) PRIVILEGED_FUNCTION {
    struct mallinfo mi = mallinfo(); // available space now managed by newlib
    return mi.fordblks + heapBytesRemaining; // plus space not yet handed to newl
}

// GetMinimumEverFree is not available in newlib's malloc implementation.
// So, no implementation is provided: size_t xPortGetMinimumEverFreeHeapSize( voi

//! No implementation needed, but stub provided in case application already calls
void vPortInitialiseBlocks( void ) PRIVILEGED_FUNCTION {};
```

Summary

Unfortunately some vendors distribute tool chains with incorrect examples of FreeRTOS/newlib.
However, with a bit of care newlib works well and safely with FreeRTOS.
Enjoy!
Best Regards, Dave

PS: newlib provides facilities for wrapping stdio functions, not covered in this article. You will want to use these, for example, if your application uses posix IO functions to read and write to a USB stick using a local FAT implementation.

PS: Hope Richard Barry and the FreeRTOS team will add `heap_useNewlib.c` to FreeRTOS ;-)

Additional References

http://www.billgatliff.com/newlib.html
http://wiki.osdev.org/Porting_Newlib
http://www.embecosm.com/appnotes/ean9/ean9-howto-newlib-1.0.html

Last Update 29-June-2020

Contact Dave Nadler:
voice USA East Coast +01 (978) 263-0097
Skype Dave.Nadler1
email Dave.Nadler@Nadler.com
Nadler & Associates Home Page